Inductive Sets of Data

Essentials of Programming Languages (Chapter 1)

I appreciate Dr. Shirin Baghoolizadeh for her helps for preparing this presentation.

Recursively Specified Data

- We introduce the basic programming tools to write interpreters, ...
- The syntax of a language is usually a nested or treelike structure
  - Recursion will be at the core of our techniques
- We must know
  - What kinds of values may occur as arguments to the procedure?
  - What kinds of values are legal for the procedure to return?

Often these sets of values are complex

Inductive Specification (top-down definition)

Powerful method for specifying a set of values

- ✓ $S \subseteq \{0, 1, 2, \dots\}$

Definition 1.1.1 *A natural number n is in S if and only if*

1. $n = 0$, *or*
2. $n - 3 \in S$.

If n is a natural number and is not a multiple of 3, then $n \notin S$.

Inductive Specification

```
in-S? : N → Bool
usage: (in-S? n) = #t if n is in S, #f otherwise
(define in-S?
  (lambda (n)
    (if (zero? n) #t
      (if (>= (- n 3) 0)
        (in-S? (- n 3))
        #f))))
```

in-S? : $N \rightarrow Bool$ is a comment, called the *contract*

Inductive Specification (bottom-up definition)

Definition 1.1.2 *Define the set S to be the smallest set contained in N and satisfying the following two properties:*

1. $0 \in S$, *and*
2. *if* $n \in S$, *then* $n + 3 \in S$.

We need "smallest set":

- Otherwise there are many sets that satisfy the remaining two conditions

Rules of Inference

$$\frac{}{0 \in S}$$

$$\frac{n \in S}{(n+3) \in S}$$

- Each entry is called a *rule of inference*, or just a *rule*
- The horizontal line is read as an "*if-then*"
- The part above the line is called the *hypothesis*
- The part below the line is called the *conclusion* or the *consequent*
- Hypotheses are connected by an implicit "and"
- A rule with no hypotheses is called an *axiom*

Rules of Inference

$$\frac{}{0 \in S}$$

$$\frac{n \in S}{(n+3) \in S}$$

- A natural number n is in S if and only if the statement "$n \in S$" can be derived from the axioms by using the rules of inference finitely many times
- This makes S the smallest set that is closed under the rules
- These definitions all say the same thing.
  - the first version: a top-down definition
  - the second version: a bottom-up definition
  - the third version: a rules-of-inference version

Inductive Specification

Definition 1.1.3 (list of integers, top-down) *A Scheme list is a* list of integers *if and only if either*

1. *it is the empty list, or*
2. *it is a pair whose car is an integer and whose cdr is a list of integers.*

Definition 1.1.4 (list of integers, bottom-up) *The set List-of-Int is the smallest set of Scheme lists satisfying the following two properties:*

1. $() \in$ *List-of-Int, and*
2. *if* $n \in$ *Int and* $l \in$ *List-of-Int, then* $(n \ . \ l) \in$ *List-of-Int.*

Rules of Inference

Definition 1.1.5 (list of integers, rules of inference)

$$() \in \textit{List-of-Int}$$

$$\frac{n \in \textit{Int} \qquad l \in \textit{List-of-Int}}{(n\ .\ l) \in \textit{List-of-Int}}$$

1. `()` is a list of integers, because of property 1 of definition 1.1.4 or the first rule of definition 1.1.5.

2. `(14 . ())` is a list of integers, because of property 2 of definition 1.1.4, since `14` is an integer and `()` is a list of integers. We can also write this as an instance of the second rule for *List-of-Int*.

$$\frac{14 \in \textit{Int} \qquad () \in \textit{List-of-Int}}{(14\ .\ ()) \in \textit{List-of-Int}}$$

Rules of Inference

3. `(3 . (14 . ()))` is a list of integers, because of property 2, since `3` is an integer and `(14 . ())` is a list of integers. We can write this as another instance of the second rule for *List-of-Int*.

$$\frac{\texttt{3} \in \mathit{Int} \qquad \texttt{(14 . ())} \in \mathit{List\text{-}of\text{-}Int}}{\texttt{(3 . (14 . ()))} \in \mathit{List\text{-}of\text{-}Int}}$$

4. `(-7 . (3 . (14 . ())))` is a list of integers, because of property 2, since `-7` is a integer and `(3 . (14 . ()))` is a list of integers. Once more we can write this as an instance of the second rule for *List-of-Int*.

$$\frac{\texttt{-7} \in \mathit{Int} \qquad \texttt{(3 . (14 . ()))} \in \mathit{List\text{-}of\text{-}Int}}{\texttt{(-7 . (3 . (14 . ())))} \in \mathit{List\text{-}of\text{-}Int}}$$

5. Nothing is a list of integers unless it is built in this fashion.

Rules of Inference

We can combine the rules to get a picture of the entire chain of reasoning, called a *derivation* or *deduction tree*

$$\dfrac{\dfrac{}{\texttt{-7} \in N} \quad \dfrac{\dfrac{}{\texttt{3} \in N} \quad \dfrac{\dfrac{}{\texttt{14} \in N} \quad \dfrac{}{() \in \textit{List-of-Int}}}{\texttt{(14 . ())} \in \textit{List-of-Int}}}{\texttt{(3 . (14 . ()))} \in \textit{List-of-Int}}}{\texttt{(-7 . (3 . (14 . ())))} \in \textit{List-of-Int}}$$

Defining Sets Using Grammars

$$\textit{List-of-Int} ::= (\,)$$
$$\textit{List-of-Int} ::= (\textit{Int} \;.\; \textit{List-of-Int})$$

- In this definition, we have:
  - Nonterminal Symbols
  - Terminal Symbols
  - Productions

Defining Sets Using Grammars

$$\begin{aligned} \textit{List-of-Int} &::= (\,) \\ &::= (\textit{Int} \;.\; \textit{List-of-Int}) \end{aligned}$$

$$\begin{aligned} \textit{List-of-Int} &::= (\,) \\ \textit{List-of-Int} &::= (\textit{Int} \;.\; \textit{List-of-Int}) \end{aligned}$$

$$\textit{List-of-Int} ::= (\,) \mid (\textit{Int} \;.\; \textit{List-of-Int})$$

$$\textit{List-of-Int} ::= (\{\textit{Int}\}^*)$$

Defining Sets Using Grammars

- *Kleene star*
  - expressed by $\{\ldots\}^*$
  - a sequence of zero or more instances of whatever appears between the braces
  - a sequence with finite length

- *Kleene plus*
  - expressed by $\{\ldots\}^+$
  - a sequence of one or more instances

- *Separated list*
  - expressed by $\{\ldots\}^{*(C)}$

Defining Sets Using Grammars

For example, $\{Int\}^{*(,)}$ includes the strings

```
8
14, 12
7, 3, 14, 16
```

and $\{Int\}^{*(;)}$ includes the strings

```
8
14; 12
7; 3; 14; 16
```

Defining Sets Using Grammars

A *syntactic derivation* may be used to show that a given data value is a member of the set

- starts with the nonterminal corresponding to the set
- at each step, a nonterminal is replaced

List-of-Int
⇒ (*Int* . *List-of-Int*)
⇒ (14 . *List-of-Int*)
⇒ (14 . ())

List-of-Int
⇒ (*Int* . *List-of-Int*)
⇒ (*Int* . ())
⇒ (14 . ())

Other Useful Sets

Definition 1.1.6 (s-list, s-exp)

$$\textit{S-list} ::= (\{\textit{S-exp}\}^*)$$
$$\textit{S-exp} ::= \textit{Symbol} \mid \textit{S-list}$$

Definition 1.1.7 (binary tree)

$$\textit{Bintree} ::= \textit{Int} \mid (\textit{Symbol}\ \ \textit{Bintree}\ \ \textit{Bintree})$$

Here are some examples of such trees:

```
1
2
(foo 1 2)
(bar 1 (foo 1 2))
(baz
  (bar 1 (foo 1 2))
  (biz 4 5))
```

Other Useful Sets

The *lambda calculus* is a simple language that consists only of:

- variable references
- procedures that take a single argument
- procedure calls

Definition 1.1.8 (lambda expression)

$$
\begin{aligned}
\mathit{LcExp} &::= \mathit{Identifier} \\
&::= \texttt{(lambda (}\mathit{Identifier}\texttt{)}\ \mathit{LcExp}\texttt{)} \\
&::= \texttt{(}\mathit{LcExp}\ \ \mathit{LcExp}\texttt{)}
\end{aligned}
$$

where an identifier is any symbol other than `lambda`.

Lambda Calculus

The identifier in the second production is called the *bound variable*

- It binds or captures any occurrences of the variable in the body
- Any occurrence of that variable in the body refers to this one

```
(lambda (x) (+ x 5))
```

```
((lambda (x) (+ x 5)) (- x 7))
```

Context-free Grammars

- These grammars are said to be *context-free*
  - Their rules may be applied in any context
  - Sometimes this is not restrictive enough

Binary-search-tree ::= `()` | `(`*Int Binary-search-tree Binary-search-tree*`)`

- Binary search trees:
  - All the keys in the left subtree are less than (or equal to) the key in the current node
  - All the keys in the right subtree are greater than the key in the current node

- To determine whether a particular production can be applied in a particular syntactic derivation:
  - We have to look at the context
  - Such constraints are called context-sensitive constraints or invariants

Context-sensitive Constraints

- Context-sensitive constraints also arise when specifying the syntax of programming languages
  - In many languages every variable must be declared
  - This constraint on the use of variables is sensitive to the context

- Formal methods can be used to specify context-sensitive constraints, but these methods are far more complicated

- The usual approach is first to specify a context-free grammar. Context-sensitive constraints are then added using other methods

Induction

Theorem 1.1.1 *Let t be a binary tree, as defined in definition 1.1.7. Then t contains an odd number of nodes.*

The key to the proof is that the substructures of a tree *t* are always smaller than *t* itself. This pattern of proof is called *structural induction*.

Proof by Structural Induction

To prove that a proposition IH(s) is true for all structures s, prove the following:

1. *IH is true on simple structures (those without substructures).*
2. *If IH is true on the substructures of s, then it is true on s itself.*

Deriving Recursive Programs

The Smaller-Subproblem Principle

If we can reduce a problem to a smaller subproblem, we can call the procedure that solves the problem to solve the subproblem.

nth-element

Procedure *list-ref* takes a list, *lst* , and a zero-based index, *n*, and returns element number *n* of *lst*

```
> (list-ref '(a b c) 1)
b
```

nth-element : *List* × *Int* → *SchemeVal*
usage: (nth-element *lst n*) = the *n*-th element of *lst*

```
(define nth-element
  (lambda (lst n)
    (if (null? lst)
      (report-list-too-short n)
      (if (zero? n)
        (car lst)
        (nth-element (cdr lst) (- n 1))))))

(define report-list-too-short
  (lambda (n)
    (eopl:error 'nth-element
      "List too short by ~s elements.~%" (+ n 1))))
```

List ::= () | (*Scheme value* . *List*)

```
  (nth-element '(a b c d e) 3)
= (nth-element   '(b c d e) 2)
= (nth-element     '(c d e) 1)
= (nth-element       '(d e) 0)
= d
```

remove-first

The procedure *remove-first* should take two arguments: a symbol, *s*, and a list of symbols, *los*

The first occurrence of the symbol *s* should be removed

remove-first : *Sym* × *Listof(Sym)* → *Listof(Sym)*

```
(define remove-first
  (lambda (s los)
    (if (null? los)
      '()
      (if (eqv? (car los) s)
        (cdr los)
        (cons (car los) (remove-first s (cdr los)))))))
```

List-of-Symbol ::= `()` | `(`*Symbol* `.` *List-of-Symbol*`)`

```
> (remove-first 'a '(a b c))
(b c)
> (remove-first 'b '(e f g))
(e f g)
> (remove-first 'a4 '(c1 a4 c1 a4))
(c1 c1 a4)
> (remove-first 'x '())
()
```

occurs-free?

A variable *occurs free* in an expression *exp* if it has some occurrence in *exp* that is not inside some lambda binding of the same variable

```
> (occurs-free? 'x 'x)
#t
> (occurs-free? 'x 'y)
#f
> (occurs-free? 'x '(lambda (x) (x y)))
#f
> (occurs-free? 'x '(lambda (y) (x y)))
#t
> (occurs-free? 'x '((lambda (x) x) (x y)))
#t
> (occurs-free? 'x '(lambda (y) (lambda (z) (x (y z)))))
#t
```

LcExp ::= *Identifier*
::= (`lambda` (*Identifier*) *LcExp*)
::= (*LcExp* *LcExp*)

occurs-free?

We can summarize these cases in the rules:

- If the expression e is a variable, then the variable x occurs free in e if and only if x is the same as e.
- If the expression e is of the form `(lambda (`y`)` e'`)`, then the variable x occurs free in e if and only if y is different from x and x occurs free in e'.
- If the expression e is of the form `(`e_1 e_2`)`, then x occurs free in e if and only if it occurs free in e_1 or e_2. Here, we use "or" to mean *inclusive or*, meaning that this includes the possibility that x occurs free in both e_1 and e_2. We will generally use "or" in this sense.

occurs-free?

```
occurs-free? : Sym × LcExp → Bool
usage:    returns #t if the symbol var occurs free
          in exp, otherwise returns #f.
(define occurs-free?
  (lambda (var exp)
    (cond
      ((symbol? exp) (eqv? var exp))
      ((eqv? (car exp) 'lambda)
       (and
         (not (eqv? var (car (cadr exp))))
         (occurs-free? var (caddr exp))))
      (else
        (or
          (occurs-free? var (car exp))
          (occurs-free? var (cadr exp)))))))
```

subst

The procedure *subst* should take three arguments: two symbols, *new* and *old*, and an s-list, *slist*.

subst : $Sym \times Sym \times S\text{-}list \rightarrow S\text{-}list$

```
(define subst
  (lambda (new old slist)
    (if (null? slist)
      '()
      (cons
        (subst-in-s-exp new old (car slist))
        (subst new old (cdr slist))))))
```

subst-in-s-exp : $Sym \times Sym \times S\text{-}exp \rightarrow S\text{-}exp$

```
(define subst-in-s-exp
  (lambda (new old sexp)
    (if (symbol? sexp)
      (if (eqv? sexp old) new sexp)
      (subst new old sexp))))
```

$S\text{-}list ::= (\{S\text{-}exp\}^*)$
$S\text{-}exp ::= Symbol \mid S\text{-}list$

$S\text{-}list ::= ()$
$::= (S\text{-}exp \ . \ S\text{-}list)$
$S\text{-}exp ::= Symbol \mid S\text{-}list$

```
> (subst 'a 'b '((b c) (b () d)))
((a c) (a () d))
```

Follow the Grammar!

- Write one procedure for each nonterminal in the grammar. The procedure will be responsible for handling the data corresponding to that nonterminal, and nothing else.
- In each procedure, write one alternative for each production corresponding to that nonterminal. You may need additional case structure, but this will get you started. For each nonterminal that appears in the right-hand side, write a recursive call to the procedure for that nonterminal.

Auxiliary Procedures

Procedure *number-elements* should take any list (v_0 v_1 v_2 ...) and return the list ((0 v_0) (1 v_1) (2 v_2) ...))

number-elements-from : $Listof(SchemeVal) \times Int \rightarrow Listof(List(Int, SchemeVal))$

usage: (number-elements-from '(v_0 v_1 v_2 ...) n)
= ((n v_0) ($n+1$ v_1) ($n+2$ v_2) ...)

```
(define number-elements-from
  (lambda (lst n)
    (if (null? lst) '()
      (cons
        (list n (car lst))
        (number-elements-from (cdr lst) (+ n 1))))))
```

context argument

number-elements : $List \rightarrow Listof(List(Int, SchemeVal))$

```
(define number-elements
  (lambda (lst)
    (number-elements-from lst 0)))
```

Vectors

- Vectors are heterogeneous structures
- It occupies less space than a list of the same length
- The length of a vector is the number of elements
  - fixed when the vector is created
- The first element in a vector is indexed by zero
- Vectors are written using the notation #(object ...)

```
#(0 (2 2 2 2) "Anna")
```

Summing All the Values in a Vector

$$\sum_{i=0}^{i=length(v)-1} v_i$$

where v is a vector of integers. We generalize it by turning the upper bound into a parameter n, so that the new task is to compute

$$\sum_{i=0}^{i=n} v_i$$

where $0 \leq n < length(v)$.

Summing All the Values in a Vector

partial-vector-sum : $Vectorof(Int) \times Int \rightarrow Int$
usage: if $0 \leq n < length(v)$, then

(partial-vector-sum v n) = $\sum_{i=0}^{i=n} v_i$

```
(define partial-vector-sum
  (lambda (v n)
    (if (zero? n)
      (vector-ref v 0)
      (+ (vector-ref v n)
         (partial-vector-sum v (- n 1))))))
```

Summing All the Values in a Vector

vector-sum : $Vectorof(Int) \rightarrow Int$

usage: (vector-sum v) = $\sum_{i=0}^{i=length(v)-1} v_i$

```
(define vector-sum
  (lambda (v)
    (let ((n (vector-length v)))
      (if (zero? n)
          0
          (partial-vector-sum v (- n 1)))))))
```